AS_SeqAnalysis

Matthew Taliaferro

6/26/2020

```
library(reticulate)
#use_python('/cloud/project/miniconda/bin/python')
use_python('/Users/mtaliaferro/miniconda2/envs/three/bin/python')
```

```
library(tidyverse)
```

Overview

Thus far we have used `STAR` and `rMATS` to analyze RNAseq data from an experiment where a specific splicing factor, RBFOX2, was knocked down. We have been able to compile a list of alternative exons whose inclusion changes, either positively or negatively, upon loss of RBFOX2. This implies that the inclusion of these exons may be regulated by RBFOX2.

RBFOX2 regulates alternative exon through binding to transcripts near exons whose inclusion it will regulate. RBFOX2, like many RBPs, has a particular RNA sequence that it prefers to recognize and bind. Take this two thoughts together, it follows then that the sequence surrounding RBFOX2-sensitive exons should be enriched for RBFOX2 binding sites relative to the sequence surrounding RBFOX2-insensitive exons. By comparing these two sequence sets, we might be able to learn something about what the sequence that RBFOX2 likes to bind is. Put more simply, we can use genomics approaches to understand something about the biochemical and biophysical properties of RBFOX2.

Get regulated events

With this strategy, the first thing we will need to do is figure out which exons are sensitive to the loss of RBFOX2. For the purposes of this class, we will assume that RBFOX2 only **promotes** exon inclusion. Note that this is not a particularly great assumption. Many RBPs will promote inclusion or exclusion depending on whether they bind the transcript relative to the exon. However, for teaching purposes, removing exons that are repressed by RBFOX2 will cut the number of comparisons we need to do in half.

So, if RBFOX2 acts to **promote** inclusion of an exon, then its inclusion (PSI value) should go down in the RBFOX2 knockdown samples. That means those exons will have **negative** delta PSI values (delta PSI was defined as RBFOX2kd - Controlkd)

Let's get a list of those events, applying the read coverage filters that we did last time.

```
rmats.filtered <- read.table('../data/rMATS/RBFOX2kd/rMATSouts/SE.MATS.JC.txt', header = T) %>%
  #Split the replicate read counts that are separated by commas into different columns
  separate(., col = IJC_SAMPLE_1, into = c('IJC_S1R1', 'IJC_S1R2', 'IJC_S1R3', 'IJC_S1R4'), sep = ',', 
  separate(., col = SJC_SAMPLE_1, into = c('SJC_S1R1', 'SJC_S1R2', 'SJC_S1R3', 'SJC_S1R4'), sep = ',', 
  separate(., col = IJC_SAMPLE_2, into = c('IJC_S2R1', 'IJC_S2R2', 'IJC_S2R3', 'IJC_S2R4'), sep = ',', 
  separate(., col = SJC_SAMPLE_2, into = c('SJC_S2R1', 'SJC_S2R2', 'SJC_S2R3', 'SJC_S2R4'), sep = ',', 
  #Calculate read counts per exon per sample
  mutate(., S1R1counts = IJC_S1R1 + SJC_S1R1) %>%
```

```
  mutate(., S1R2counts = IJC_S1R2 + SJC_S1R2) %>%
  mutate(., S1R3counts = IJC_S1R3 + SJC_S1R3) %>%
  mutate(., S1R4counts = IJC_S1R4 + SJC_S1R4) %>%
  mutate(., S2R1counts = IJC_S2R1 + SJC_S2R1) %>%
  mutate(., S2R2counts = IJC_S2R2 + SJC_S2R2) %>%
  mutate(., S2R3counts = IJC_S2R3 + SJC_S2R3) %>%
  mutate(., S2R4counts = IJC_S2R4 + SJC_S2R4) %>%
  #Filter on read counts
  filter(., S1R1counts >= 20 & S1R2counts >= 20 & S1R3counts >= 20 & S1R4counts >= 20 &
           S2R1counts >= 20 & S2R2counts >= 20 & S2R3counts >= 20 & S2R4counts >= 20) %>%
  #Get rid of extraneous columns
  dplyr::select(., c(geneSymbol, chr, strand, exonStart_0base, exonEnd, FDR, IncLevelDifference))
```

OK these are all the events that pass read coverage filters. What we need to do now is make two different tables: one for the exons that are signficantly *less* included upon RBFOX2 knockdown (FDR $<$ 0.05 & IncLevelDifference $<$ -0.05) and one for exons that are insensitive to RBFOX2 knockdown (FDR $>=$ 0.05).

```
psis.sensitive <- filter(rmats.filtered, FDR < 0.05 & IncLevelDifference < 0)
psis.insensitive <- filter(rmats.filtered, FDR >= 0.5)

nrow(psis.sensitive)
```

```
## [1] 114
```

```
nrow(psis.insensitive)
```

```
## [1] 4087
```

```
head(psis.sensitive)
```

```
##   geneSymbol   chr strand exonStart_0base  exonEnd          FDR
## 1        Mff  chr1      +        82741817 82741976 0.0025282307
## 2      Rps24 chr14      +        24495429 24495449 0.0000000000
## 3      Egfl7  chr2      +        26590379 26590495 0.0073345471
## 4      Scnm1  chr3      -        95130163 95130358 0.0003716722
## 5     Fam49b chr15      -        63956599 63956682 0.0003471414
## 6     Fam49b chr15      -        63956599 63956682 0.0003762508
##   IncLevelDifference
## 1             -0.060
## 2             -0.141
## 3             -0.143
## 4             -0.148
## 5             -0.100
## 6             -0.155
```

Get sequences

Alright, now we have the two groups of exons we want, but if we are going to find RNA sequence enriched in one or the other, we actually need the sequences that are flanking the exons. How can we do that? Specifically, these are the sequences that we want:

A given nucleotide can be defined in the genome with 3 parameters: chromosome, coordinate, and strand. Luckily, rMATS records these data for the alternative exon as well as it's two neighbor exons, upstream and downstream. Consider the first event above from the gene Mff. The alternative exon begins at corrdiate 82741817 in chr1 and is on the + strand. The exon extends until coordinate 82741976. OK so how can we go from this data to actual sequence, you know, As and Cs and Gs and Us and such?

There are many ways to do this, but we are going to use our old friend `PYTHON`.

When we were learning python in the bootcamp, we talked about slicing strings using indicies.

```
#A very short chrosomsome
chr = 'ACTGATCGATCATCGATCGATCGATCGATCGTAGTAGTAGTCGTACGATCG'
#My sequence of interest begins at (0-based) 4 and goes up to (but doesn't include!!) 12
myseq = chr[4:12]
print(myseq)
```

```
## ATCGATCA
```

So you can see what's going on here, but imagine that we had more than one chromosome, which all interesting organisms do (sorry E. coli). How would we deal with that? Here our old friend the dictionary will come to the rescue. The keys in our dictionary will be chromosome names and their values will be the sequence of the chromosome.

```
#A very short chrosomsome
chr1seq = 'ACTGATCGATCATCGATCGATCGATCGATCGTAGTAGTAGTCGTACGATCG'
chr2seq = 'TGATCGATCGATCGATCGATCGAGCAGCTACTATCATCGATCGATCGCCAA'

genome = {}
genome['chr1'] = chr1seq
genome['chr2'] = chr2seq

#My sequence of interest begins at (0-based) 4 of chr1 and goes up to (but doesn't include!!) 12
myseq = genome['chr1'][4:12]
print(myseq)
```

```
## ATCGATCA
```

The sequences that we want to analyze are the 150 nt that flank the alternative exon. Let's write files that have information on where those sequences are.

```
psis.sensitive %>%
  dplyr::select(., -c(FDR, IncLevelDifference)) %>%
  write.table(., file = '../data/sequences/sensexons.coords.txt', sep = '\t', row.names = F, col.names =

psis.insensitive %>%
```

```
  dplyr::select(., -c(FDR, IncLevelDifference)) %>%
  write.table(., file = '../data/sequences/insensexons.coords.txt', sep = '\t', row.names = F, col.names
```

Now we are ready to take those coordinates and retrieve their sequences. The `biopython` library has a set of useful functions here. You can give it a genome sequence, and it will make a dictionary that is of the structure we described above where keys are chromosome names and values are sequences. For simplicity here, we will pretend that the entire of the genome is chromosome 19.

```
from Bio import SeqIO

genomefasta = '../data/dummySTAR/chr19.fasta'
#Make a Biopython 'fasta object' of the genome
genomefasta_obj = SeqIO.parse(open(genomefasta, 'r'), 'fasta')
#Make a dictionary of the 'genome'
seq_dict = SeqIO.to_dict(genomefasta_obj)


#Now let's read in the coordinate files we made earlier.
#For each file, we will make two fastas: one of the upstream intronic 150 nt and one of the downstream
with open('../data/sequences/sensexons.coords.txt', 'r') as coordfh, open('../data/sequences/sensexons.
  for line in coordfh:
    #Remove trailing newline characters and turn each line into a list
    line = line.strip().split('\t')
    chrm = line[1]
    strand = line[2]
    exonstart = int(line[3])
    exonstop = int(line[4])
    seqid = ('_').join(line)

    #If this exon wasn't on chr19, skip it because we only have genome sequences for chr19
    if chrm != 'chr19':
      continue

    #If this is a positive strand gene, things are pretty straightfoward
    if strand == '+':
      upstreamintstart = exonstart - 150
      upstreamintend = exonstart
      seq = seq_dict[chrm].seq[upstreamintstart : upstreamintend].transcribe()
    #If it's a negative strand gene the upstream intron (in the RNA sense) is actually after this intro
    #AND we need to take the reverse complement
    elif strand == '-':
      upstreamintstart = exonstop
      upstreamintend = exonstop + 150
      seq = seq_dict[chrm].seq[upstreamintstart : upstreamintend].reverse_complement().transcribe() #bi

    #Write sequence in fasta format
    outfh.write ('>' + seqid + '\n' + str(seq) + '\n')
```

```
## 183
## 182
## 183
```

That was meant as a small vignette to get you introduced into how we might go from genome coordinates to nucleotide sequence. The chunk below will make upstream and downstream sequence files for all exons, not just those on chr19. Do not run it here, as it will not work because it is missing the genome fasta file (it's too

big to deal with easily in this class).

```
from Bio import SeqIO
import gzip

genomefasta = '/Users/mtaliaferro/Desktop/Annotations/mm10/GencodeM17/GRCm38.primary_assembly.genome.fa

seq_dict = SeqIO.to_dict(SeqIO.parse(gzip.open(genomefasta, 'rt'), 'fasta'))

def coordstoseq(coords, upstreamseqfile, downstreamseqfile):
  with open(coords, 'r') as coordfh, open(upstreamseqfile, 'w') as upstreamoutfh, open(downstreamseqfil
      for line in coordfh:
        #Remove trailing newline characters and turn each line into a list
        line = line.strip().split('\t')
        chrm = line[1]
        strand = line[2]
        exonstart = int(line[3])
        exonstop = int(line[4])
        seqid = ('_').join(line)

        #If this is a positive strand gene, things are pretty straightfoward
        if strand == '+':
          upstreamintstart = exonstart - 150
          upstreamintend = exonstart
          downstreamintstart = exonstop
          downstreamintend = exonstop + 150
          upstreamseq = seq_dict[chrm].seq[upstreamintstart : upstreamintend].transcribe()
          downstreamseq = seq_dict[chrm].seq[downstreamintstart : downstreamintend].transcribe()
        #If it's a negative strand gene the upstream intron (in the RNA sense) is actually after this i
        #AND we need to take the reverse complement
        elif strand == '-':
          upstreamintstart = exonstop
          upstreamintend = exonstop + 150
          downstreamintstart = exonstart - 150
          downstreamintend = exonstart
          upstreamseq = seq_dict[chrm].seq[upstreamintstart : upstreamintend].reverse_complement().trans
          downstreamseq = seq_dict[chrm].seq[downstreamintstart : downstreamintend].reverse_complement()

        #Write sequence in fasta format
        upstreamoutfh.write ('>' + seqid + '\n' + str(upstreamseq) + '\n')
        downstreamoutfh.write ('>' + seqid + '\n' + str(downstreamseq) + '\n')

#sens exons
coordstoseq('../data/sequences/sensexons.coords.txt','../data/sequences/sensexons.upstream.fa','../data/

#insens exons
coordstoseq('../data/sequences/insensexons.coords.txt','../data/sequences/insensexons.upstream.fa','../d
```

Counting kmers

Now that we have our sequences, we want to ask which kmers are enriched in the sensexons sequences relative to the insensexons sequences. Kmers are just nucleotide sequences of length k. Often in these types of analyses, we will look for the enrichment of 5mers or 6mers (k = 5 or 6). Here, we will look at 5mers.

So what we want to do is essentially ask, for every possible 5mer, what is the density of that 5mer in the sens sequences and how does it compare to the density in the insens sequences? Another way to ask that is "for all of the kmers in the sequences, what fraction of them are kmer X"? So we need to have a way to look at a sequence and count how many times each kmer occurs in that sequence. Luckily, we can do that fairly easily with `python`.

```
testseq = 'AUUAGCUAGCUAGCGACGCAGUACGUACGUAGCUAGCUAGCUAGUAUGCAUGAUGCUGACUG'

#All we need to do is take a window that is 5 nt wide and slide it along the sequence one nt at a time,
kmercounts = {} #{kmer : number of times we observe that kmer}

k = 5

for i in range(len(testseq) - k + 1):
  kmer = testseq[i : i+k]
  print(kmer)
  #If we haven't seen this kmer before, its count is 1
  if kmer not in kmercounts:
    kmercounts[kmer] = 1
  #If we have seen this kmer before, add one to its count
  elif kmer in kmercounts:
    kmercounts[kmer] +=1
```

```
## AUUAG
## UUAGC
## UAGCU
## AGCUA
## GCUAG
## CUAGC
## UAGCU
## AGCUA
## GCUAG
## CUAGC
## UAGCG
## AGCGA
## GCGAC
## CGACG
## GACGC
## ACGCA
## CGCAG
## GCAGU
## CAGUA
## AGUAC
## GUACG
## UACGU
## ACGUA
## CGUAC
## GUACG
## UACGU
## ACGUA
## CGUAG
## GUAGC
## UAGCU
```

```
## AGCUA
## GCUAG
## CUAGC
## UAGCU
## AGCUA
## GCUAG
## CUAGC
## UAGCU
## AGCUA
## GCUAG
## CUAGU
## UAGUA
## AGUAU
## GUAUG
## UAUGC
## AUGCA
## UGCAU
## GCAUG
## CAUGA
## AUGAU
## UGAUG
## GAUGC
## AUGCU
## UGCUG
## GCUGA
## CUGAC
## UGACU
## GACUG
```

```
print(kmercounts)
```

```
## {'AUUAG': 1, 'UUAGC': 1, 'UAGCU': 5, 'AGCUA': 5, 'GCUAG': 5, 'CUAGC': 4, 'UAGCG': 1, 'AGCGA': 1, 'GC
```

OK all we need to do now is find a way to take in a fasta of sequences instead of an individual 'testseq'.

```
from Bio import SeqIO
from itertools import product

#Make a list of all possible kmers
nucleotides = ['A', 'G', 'C', 'U']
allkmers = [''.join(kmer) for kmer in product(nucleotides, repeat=5)]

def countkmers(fasta, outfile, k):
  k = int(k)
  kmercounts = {} #{kmer : number of times we observe that kmer}
  with open(fasta, 'r') as fastafh:
    #For every sequence in this file
    for record in SeqIO.parse(fastafh, 'fasta'):
      seq = str(record.seq)
      #Count kmers
      for i in range(len(seq) - k + 1):
        kmer = seq[i : i+k]
        #If we haven't seen this kmer before, its count is 1
        if kmer not in kmercounts:
          kmercounts[kmer] = 1
        #If we have seen this kmer before, add one to its count
```

```
        elif kmer in kmercounts:
          kmercounts[kmer] +=1

  #OK so by the time we get to here we will have a dictionary of all kmers that we saw and how many tim
  #If we didn't see a kmer, lets add it in and say we saw it 0 times
  for kmer in allkmers:
    if kmer not in kmercounts:
      kmercounts[kmer] = 0

  #Alright so now, for each kmer, I want to write how many times I saw that kmer and how many times I s
  with open(outfile, 'w') as outfh:
    totalkmercounts = sum(kmercounts.values())
    outfh.write('kmer' + '\t' + 'kmercount' + '\t' + 'otherkmerscount' + '\n')
    for kmer in kmercounts:
      kmercount = kmercounts[kmer]
      otherkmerscount = totalkmercounts - kmercount
      outfh.write(kmer + '\t' + str(kmercount) + '\t' + str(otherkmerscount) + '\n')


#Insensitve exons, upstream
countkmers('../data/sequences/insensexons.upstream.fa', '../data/sequences/insensexons.upstream.kmercou

#Insensitve exons, downstream
countkmers('../data/sequences/insensexons.downstream.fa', '../data/sequences/insensexons.downstream.kme

#Sensitve exons, upstream
countkmers('../data/sequences/sensexons.upstream.fa', '../data/sequences/sensexons.upstream.kmercounts.

#Sensitve exons, downstream
countkmers('../data/sequences/sensexons.downstream.fa', '../data/sequences/sensexons.downstream.kmercou
```

OK you now have four files, one for each region (upstream and downstream) in each sequence set (sens and insens). What we would like to know is which kmers are significantly enriched across sequence sets within a region (e.g. upstream sens vs. upstream insens). We have all the data necessary to do that. It's up to you in the exercises to figure out which of the 1024 possible 5mers are enriched in the sequences surrounding RBFOX2-sensitive exons.